Automated Program Verification and Testing
15414/15614 Fall 2016
Lecture 26:
Counterexamples & Abstraction Refinement

Matt Fredrikson
mfredrik@cs.cmu.edu

December 6, 2016
**Key Idea**: Approximate system so that a given property is preserved
**Key Idea:** Approximate system so that a given property is preserved

More precisely, given KS $M$ and $\phi$, we want $\hat{M}$ such that

$$\hat{M} \models \phi \Rightarrow M \models \phi$$
**Key Idea**: Approximate system so that a given property is preserved

More precisely, given KS $M$ and $\phi$, we want $\hat{M}$ such that

$$\hat{M} \models \phi \Rightarrow M \models \phi$$

We’ll see how to build a **conservative overapproximation** of $M$
**Key Idea:** Approximate system so that a given property is preserved

More precisely, given KS $\mathcal{M}$ and $\phi$, we want $\hat{\mathcal{M}}$ such that

$$\hat{\mathcal{M}} \models \phi \Rightarrow \mathcal{M} \models \phi$$

We’ll see how to build a conservative overapproximation of $\mathcal{M}$

- Every trace of $\mathcal{M}$ is also a trace of $\hat{\mathcal{M}}$
**Key Idea**: Approximate system so that a given property is preserved

More precisely, given KS $M$ and $\phi$, we want $\hat{M}$ such that

$$\hat{M} \models \phi \Rightarrow M \models \phi$$

We’ll see how to build a **conservative overapproximation** of $M$

- Every trace of $M$ is also a trace of $\hat{M}$
- Some traces in $\hat{M}$ may not be in $M$
Key Idea: Approximate system so that a given property is preserved

More precisely, given KS $M$ and $\phi$, we want $\hat{M}$ such that

$$\hat{M} \models \phi \Rightarrow M \models \phi$$

We’ll see how to build a conservative overapproximation of $M$

- Every trace of $M$ is also a trace of $\hat{M}$
- Some traces in $\hat{M}$ may not be in $M$

This preserves safety properties: if $\hat{M}$ verifies, so will $M$
**Key Idea:** Approximate system so that a given property is preserved

More precisely, given KS $M$ and $\phi$, we want $\hat{M}$ such that

$$\hat{M} \models \phi \Rightarrow M \models \phi$$

We’ll see how to build a **conservative overapproximation** of $M$

- Every trace of $M$ is also a trace of $\hat{M}$
- Some traces in $\hat{M}$ may not be in $M$

This preserves safety properties: if $\hat{M}$ verifies, so will $M$

But it might introduce **spurious counterexamples**
How do we know which abstraction to use?

Idea:
- Only track predicates on program's data state
  - Predicates relevant to the property, control flow
  - Each state in the transition maps to a vector of predicate values

We're given: set of predicates $E = \phi_1; \ldots; \phi_n$

Define abstraction function:
$$\text{Env} \mapsto f_0; 1$$

$$(\ell; (\phi_1(\cdot); \ldots; \phi_n(\cdot)))$$

Intuitively:
ranges over conjunctions of $\phi_i$;
The states in our abstraction will be:
$S = \text{Loc } f_0; 1$
How do we know which abstraction to use?

Idea: Only track **predicates** on program’s data state

- Predicates relevant to the property, control flow
- Each state in the transition maps to a vector of predicate values
How do we know which abstraction to use?

**Idea**: Only track **predicates** on program’s data state
- Predicates relevant to the property, control flow
- Each state in the transition maps to a vector of predicate values

We’re given: set of predicates $E = \{\phi_1, \ldots, \phi_n\}$

Define **abstraction function** $\alpha : \text{Env} \mapsto \{0, 1\}^n$:

$$\alpha((\ell, \sigma)) = (\ell, (\phi_1(\sigma), \ldots, \phi_n(\sigma)))$$
How do we know which abstraction to use?

**Idea**: Only track **predicates** on program’s data state

- Predicates relevant to the property, control flow
- Each state in the transition maps to a vector of predicate values

We’re given: set of predicates $E = \{\phi_1, \ldots, \phi_n\}$

Define **abstraction function** $\alpha : \text{Env} \mapsto \{0, 1\}^n$:

$\alpha((l, \sigma)) = (l, (\phi_1(\sigma), \ldots, \phi_n(\sigma)))$

Intuitively: $\alpha$ ranges over conjunctions of $\phi_i, \neg\phi_i$
How do we know which abstraction to use?

**Idea:** Only track **predicates** on program’s data state
- Predicates relevant to the property, control flow
- Each state in the transition maps to a vector of predicate values

We’re given: set of predicates \( E = \{\phi_1, \ldots, \phi_n\} \)

Define **abstraction function** \( \alpha : \text{Env} \mapsto \{0, 1\}^n : \)
\[
\alpha((l, \sigma)) = (l, (\phi_1(\sigma), \ldots, \phi_n(\sigma)))
\]

Intuitively: \( \alpha \) ranges over conjunctions of \( \phi_i, \neg \phi_i \)

The states in our abstraction will be: \( S = \text{Loc} \times \{0, 1\}^m \)
Important:

We want an over-approximation that gives us:

\[ M_j = \phi \]

We’ll define an existential abstraction:

\[ (s_1; s_2) \sim R, \exists s_1; s_2 : \mathcal{R}(s_1; s_2) = h(s_1) = s_1 h(s_2) = s_2 \]

A transition is in the abstraction \( M \) if and only if:

1. There exist corresponding states \((s_1; s_2)\) in \( M \), where \( s_1; s_2 \) are the endpoints of a transition in \( M \)

Why is this conservative?
Important: We want an over-approximation that gives us:

\[ \hat{M} \models \phi \Rightarrow M \models \phi \]
**Important:** We want an over-approximation that gives us:

$$\hat{M} \models \phi \Rightarrow M \models \phi$$

We’ll define an **existential abstraction**:

$$(\hat{s}_1, \hat{s}_2) \in \hat{R} \iff \exists s_1, s_2. R(s_1, s_2) \land h(s_1) = \hat{s}_1 \land h(s_2) = \hat{s}_2$$

$$\hat{s} \in \hat{I} \iff \exists s. s \in I \land h(s) = \hat{s}$$

A transition is in the abstraction $\hat{M}$ if and only if:

1. There **exist** corresponding states $(s_1, s_2)$ in $M$,
2. where $s_1, s_2$ are the endpoints of a transition in $M$
Important: We want an over-approximation that gives us:

\[ \hat{M} \models \phi \Rightarrow M \models \phi \]

We’ll define an existential abstraction:

\[
(\hat{s}_1, \hat{s}_2) \in \hat{R} \iff \exists s_1, s_2. R(s_1, s_2) \land h(s_1) = \hat{s}_1 \land h(s_2) = \hat{s}_2
\]

\[
\hat{s} \in \hat{I} \iff \exists s. s \in I \land h(s) = \hat{s}
\]

A transition is in the abstraction \( \hat{M} \) if and only if:

1. There exist corresponding states \((s_1, s_2)\) in \( M \),
2. where \( s_1, s_2 \) are the endpoints of a transition in \( M \)

Why is this conservative?
Intuition: Existential Abstraction

Image Credit: Tom Henzinger, Ranjit Jhala, Rupak Majumdar
Intuition: Existential Abstraction

Image Credit: Tom Henzinger, Ranjit Jhala, Rupak Majumdar
The key issue: how do we compute transitions
The key issue: how do we compute transitions

Recall our construction of KS from program graphs:

\[
\begin{align*}
(l_1, b, l_2) \in T & \quad \langle b, \sigma_1 \rangle \downarrow_b \text{true} \quad \langle C(l_1), \sigma_1 \rangle \downarrow \sigma_2 \\
([l_1, \sigma_1], [l_2, \sigma_2]) & \in R
\end{align*}
\]
The key issue: how do we compute transitions

Recall our construction of KS from program graphs:

\[
(\ell_1, b, \ell_2) \in T \quad \langle b, \sigma_1 \rangle \Downarrow_b \text{true} \quad \langle C(\ell_1), \sigma_1 \rangle \Downarrow \sigma_2
\]

\[
([\ell_1, \sigma_1], [\ell_2, \sigma_2]) \in R
\]

We don’t have concrete states \( \sigma \) to work with anymore
The key issue: how do we compute transitions

Recall our construction of KS from program graphs:

\[
(\ell_1, b, \ell_2) \in T \quad \langle b, \sigma_1 \rangle \Downarrow_b \text{true} \quad \langle C(\ell_1), \sigma_1 \rangle \Downarrow \sigma_2
\]

\[
([\ell_1, \sigma_1], [\ell_2, \sigma_2]) \in R
\]

We don’t have concrete states $\sigma$ to work with anymore

Just predicates.
The key issue: how do we compute transitions

Recall our construction of KS from program graphs:

\[(\ell_1, b, \ell_2) \in T \quad \langle b, \sigma_1 \rangle \Downarrow_b \text{true} \quad \langle C(\ell_1), \sigma_1 \rangle \Downarrow \sigma_2 \]

\[([\ell_1, \sigma_1], [\ell_2, \sigma_2]) \in R\]

We don’t have concrete states \(\sigma\) to work with anymore

Just predicates. **Idea**: Use predicate transformers
Given $E = \{\phi_1, \ldots, \phi_n\}$, let $\text{Pred}(\phi, E)$:

- The **weakest** DNF over $E$,
- that is at least as strong as $\phi$,
- where each clause has $n$ literals

Notice: $\text{Pred}(\phi, E) \Rightarrow \phi$

Compute this by querying SMT solver

- What’s the complexity of this?
- $O(2^n)$
- Need to query each:
  
  $$p_1 \land \cdots \land p_n \Rightarrow \phi$$

  where $p_i$ is $\phi_i$ or $\neg \phi_i$
For assignments $x := e$:
For assignments $x := e$:

1. Compute $wp(x := e, \phi)$, $wp(x := e, \neg\phi)$
Computing Transitions via Strengthening

For assignments $x := e$:

1. Compute $wp(x := e, \phi)$, $wp(x := e, \neg \phi)$
2. Strengthen them: $\text{Pred}(wp(x := e, \phi), E)$, $\text{Pred}(\neg wp(x := e, \phi), E)$
Computing Transitions via Strengthening

For assignments $x := e$:

1. Compute $\text{wp}(x := e, \phi)$, $\text{wp}(x := e, \neg\phi)$
2. Strengthen them: $\text{Pred}(\text{wp}(x := e, \phi), E)$, $\text{Pred}(\neg\text{wp}(x := e, \phi), E)$
3. If state implies $\text{Pred}(\text{wp}(x := e, \phi), E)$, draw an edge to $\phi$
4. If state implies $\text{Pred}(\neg\text{wp}(x := e, \phi), E)$, draw an edge to $\phi$
5. If neither implication holds, draw an edge to both $\ell_0$ and $\ell_1$: $x := x + 1$, $\text{skip}$

E = $\{x = y | \{z\}}$
For assignments $x := e$:

1. Compute $wp(x := e, \phi), wp(x := e, \neg \phi)$
2. Strengthen them: $\text{Pred}(wp(x := e, \phi), E), \text{Pred}(\neg wp(x := e, \phi), E)$
3. If state implies $\text{Pred}(wp(x := e, \phi), E)$, draw an edge to $\phi$
4. If state implies $\text{Pred}(\neg wp(x := e, \phi), E)$, draw an edge to $\neg \phi$
Computing Transitions via Strengthening

For assignments $x := e$:

1. Compute $wp(x := e, \phi), wp(x := e, \neg\phi)$
2. Strengthen them: $\text{Pred}(wp(x := e, \phi), E), \text{Pred}(\neg wp(x := e, \phi), E)$
3. If state implies $\text{Pred}(wp(x := e, \phi), E)$, draw an edge to $\phi$
4. If state implies $\text{Pred}(\neg wp(x := e, \phi), E)$, draw an edge to $\neg\phi$
5. If neither implication holds, draw an edge to both
For assignments $x := e$:

1. Compute $wp(x := e, \phi), wp(x := e, \neg\phi)$
2. Strengthen them: $\text{Pred}(wp(x := e, \phi), E), \text{Pred}(\neg wp(x := e, \phi), E)$
3. If state implies $\text{Pred}(wp(x := e, \phi), E)$, draw an edge to $\phi$
4. If state implies $\text{Pred}(\neg wp(x := e, \phi), E)$, draw an edge to $\neg\phi$
5. If neither implication holds, draw an edge to both

\[
\ell_0 : \quad x := x + 1 \\
\ell_1 : \quad \text{skip}
\]

\[
E = \{x = y\}_{p_0}
\]
For assignments $x := e$:

1. Compute $wp(x := e, \phi), \ \wp(x := e, \neg\phi)$
2. Strengthen them: $\text{Pred}(wp(x := e, \phi), E), \ \text{Pred}(\neg\wp(x := e, \phi), E)$
3. If state implies $\text{Pred}(wp(x := e, \phi), E)$, draw an edge to $\phi$
4. If state implies $\text{Pred}(\neg\wp(x := e, \phi), E)$, draw an edge to $\neg\phi$
5. If neither implication holds, draw an edge to both

\[\ell_0 : \ x := x + 1\]
\[\ell_1 : \ \text{skip}\]

\[E = \{x = y\}\]

\[p_0\]

\[\neg p_0\]
For assumptions \textbf{assume }\phi: 

1. Weaken \phi:
   \text{Pred}(\phi; E)

2. Strengthen them:
   \text{Pred}(wp(x:=e; \phi); E), \text{Pred}(wp(x:=e; \phi); E)

3. If next state implies \phi, draw an edge to it

4. If next state implies \phi, draw an edge to it
For assumptions assume $\phi$:
1. *Weaken* $\phi$: $\neg \text{Pred}(\neg \phi, E)$
Computing Transitions via Strengthening

For assumptions assume $\phi$:

1. **Weaken** $\phi$: $\neg \text{Pred}(\neg \phi, E)$

2. **Strengthen** them: $\text{Pred}(\text{wp}(x := e, \phi), E)$, $\text{Pred}(\neg \text{wp}(x := e, \phi), E)$
For assumptions \textbf{assume} $\phi$:

1. \textit{Weaken} $\phi$: $\neg \text{Pred}(\neg \phi, E)$

2. Strengthen them: $\text{Pred}(\text{wp}(x := e, \phi), E)$, $\text{Pred}(\neg \text{wp}(x := e, \phi), E)$

3. If next state implies $\neg \text{Pred}(\neg \phi, E)$, draw an edge to it
Computing Transitions via Strengthening

For assumptions **assume** \( \phi \):

1. **Weaken** \( \phi \): \( \neg \text{Pred}(\neg \phi, E) \)
2. Strengthen them: \( \text{Pred}(\text{wp}(x := e, \phi), E), \text{Pred}(\neg \text{wp}(x := e, \phi), E) \)
3. If next state implies \( \neg \text{Pred}(\neg \phi, E) \), draw an edge to it
4. If next state implies \( \neg \text{Pred}(\phi, E) \), draw an edge to it
For assumptions **assume** $\phi$:

1. **Weaken** $\phi$: $\neg\text{Pred}(-\phi, E)$
2. Strengthen them: $\text{Pred}(wp(x := e, \phi), E)$, $\text{Pred}(-wp(x := e, \phi), E)$
3. If next state implies $\neg\text{Pred}(-\phi, E)$, draw an edge to it
4. If next state implies $\neg\text{Pred}(\phi, E)$, draw an edge to it

$\ell_0 : \text{assume } x = 1$

$\ell_1 : \text{skip}$

$$E = \{ x = y \}$$

$p_0$
Computing Transitions via Strengthening

For assumptions assume $\phi$:

1. **Weaken** $\phi$: $\neg\text{Pred}(\neg\phi, E)$
2. Strengthen them: $\text{Pred}(\text{wp}(x := e, \phi), E)$, $\text{Pred}(\neg\text{wp}(x := e, \phi), E)$
3. If next state implies $\neg\text{Pred}(\neg\phi, E)$, draw an edge to it
4. If next state implies $\neg\text{Pred}(\phi, E)$, draw an edge to it

$\ell_0 : \text{assume } x = 1$

$\ell_1 : \text{skip}$

$$E = \{x = y\}$$

$\text{Pred}(\neg(x = 1), \{x = y\}) =$
For assumptions **assume** $\phi$:

1. **Weaken** $\phi$: $\neg \text{Pred}(\neg \phi, E)$
2. Strengthen them: $\text{Pred}(\text{wp}(x := e, \phi), E)$, $\text{Pred}(\neg \text{wp}(x := e, \phi), E)$
3. If next state implies $\neg \text{Pred}(\neg \phi, E)$, draw an edge to it
4. If next state implies $\neg \text{Pred}(\phi, E)$, draw an edge to it

$\ell_0 : \text{ assume } x = 1$

$\ell_1 : \text{ skip}$

$$E = \{ x = y \}$$

$\text{Pred}(\neg (x = 1), \{ x = y \}) = \text{false}$
Computing Transitions via Strengthening

For assumptions **assume** $\phi$:

1. *Weaken* $\phi$: $\neg \text{Pred}(\neg \phi, E)$
2. Strengthen them: $\text{Pred}(\wp(x := e, \phi), E)$, $\text{Pred}(\neg \wp(x := e, \phi), E)$
3. If next state implies $\neg \text{Pred}(\neg \phi, E)$, draw an edge to it
4. If next state implies $\neg \text{Pred}(\phi, E)$, draw an edge to it

$l_0$ : **assume** $x = 1$

$l_1$ : **skip**

\[ E = \{ x = y \} \]

$\text{Pred}(\neg(x = 1), \{x = y\}) = \text{false}$

$\text{Pred}(x = 1, \{x = y\}) = \text{}$
Computing Transitions via Strengthening

For assumptions assume $\phi$:

1. **Weaken** $\phi$: $\neg \text{Pred}(\neg \phi, E)$
2. Strengthen them: $\text{Pred}(\text{wp}(x := e, \phi), E)$, $\text{Pred}(\neg \text{wp}(x := e, \phi), E)$
3. If next state implies $\neg \text{Pred}(\neg \phi, E)$, draw an edge to it
4. If next state implies $\neg \text{Pred}(\phi, E)$, draw an edge to it

$\ell_0 : \text{assume } x = 1$

$\ell_1 : \text{skip}$

$$E = \{x = y\}$$

$P_0$

$\text{Pred}(\neg(x = 1), \{x = y\}) = \text{false}$

$\text{Pred}(x = 1, \{x = y\}) = \text{false}$
For assumptions assume $\phi$:

1. *Weaken* $\phi$: $\neg\text{Pred}(\neg\phi, E)$
2. Strengthen them: $\text{Pred}(\text{wp}(x := e, \phi), E)$, $\text{Pred}(\neg\text{wp}(x := e, \phi), E)$
3. If next state implies $\neg\text{Pred}(\neg\phi, E)$, draw an edge to it
4. If next state implies $\neg\text{Pred}(\phi, E)$, draw an edge to it

$p_0$: assume $x = 1$

$p_1$: skip

$E = \{x = y\}$

$\text{Pred}(\neg(x = 1), \{x = y\}) = false$

$\text{Pred}(x = 1, \{x = y\}) = false$
Example: Predicate Abstraction

\[ \ell_0 : \ i := 1; \]
\[ \ell_1 : \ \textbf{while}(0 \leq x < 1) \ {\}
\[ \ell_2 : \ i := i - 1; \]
\[ \ell_3 : \ x := x + 1; \]
\[ \} \]

Suppose we check:

\[ G (\neg \ell_0 \rightarrow 0 \leq i) \]

Using:

\[ E = \{0 \leq i\} \]

\[ p_0 \]
Example: Predicate Abstraction

\[\ell_0 : \ i := 1;\]
\[\ell_1 : \ \textbf{while}(0 \leq x < 1) \ \{\]
\[\ell_2 : \ i := i - 1;\]
\[\ell_3 : \ x := x + 1;\]
\[
\}

Suppose we check:
\[\mathcal{G} (\neg \ell_0 \rightarrow 0 \leq i)\]

Using:
\[E = \{0 \leq i\}\]
Example: Predicate Abstraction

\[ \ell_0 : \ i := 1; \]
\[ \ell_1 : \ \texttt{while}(0 \leq x < 1) \{ \]
\[ \ell_2 : \ i := i - 1; \]
\[ \ell_3 : \ x := x + 1; \]
\[ \} \]

Suppose we check:

\[ \mathbf{G} \ (\neg \ell_0 \rightarrow 0 \leq i) \]

Using:

\[ E = \{0 \leq i\} \]

\[ \ell_0 : \ p_0 \]

\[ \ell_1 : \ p_0 \]

\[ \ell_2 : \ p_0 \]

\[ \ell_3 : \ p_0 \]
Example: Predicate Abstraction

\[ \ell_0 : \ i := 1; \]
\[ \ell_1 : \ \textbf{while}(0 \leq x < 1) \{ \]
\[ \ell_2 : \ i := i - 1; \]
\[ \ell_3 : \ x := x + 1; \]
\[ \} \]

Suppose we check:

\[ \mathbf{G} (\neg \ell_0 \rightarrow 0 \leq i) \]

Using:

\[ E = \{0 \leq i\} \]

\[ p_0 \]
Example: Predicate Abstraction

\[ \ell_0 : \ i := 1; \]
\[ \ell_1 : \ \textbf{while}(0 \leq x < 1) \ { \}
\[ \ell_2 : \ i := i - 1; \]
\[ \ell_3 : \ x := x + 1; \]
\}

Suppose we check:
\[ G \ (\neg \ell_0 \rightarrow 0 \leq i) \]

Using:
\[ E = \{0 \leq i\} \]
\[ p_0 \]
Example: Predicate Abstraction

\[ \ell_0 : \; i := 1; \]
\[ \ell_1 : \; \textbf{while}(0 \leq x < 1) \{ \]
\[ \ell_2 : \; i := i - 1; \]
\[ \ell_3 : \; x := x + 1; \]
\[ \} \]

Suppose we check:
\[ \mathbf{G} \; (\neg \ell_0 \Rightarrow 0 \leq i) \]

Using:
\[ E = \{0 \leq i\} \]
\[ \{p_0\} \]
Example: Predicate Abstraction

\[ \ell_0 : \quad i := 1; \]
\[ \ell_1 : \quad\textbf{while}(0 \leq x < 1) \{ \]
\[ \ell_2 : \quad i := i - 1; \]
\[ \ell_3 : \quad x := x + 1; \]
\[ \} \]

Suppose we check:
\[ G (\neg \ell_0 \rightarrow 0 \leq i) \]

Using:
\[ E = \{0 \leq i\} \]

\[ p_0 \]
Example: Predicate Abstraction

\[ \ell_0 : \ i := 1; \]
\[ \ell_1 : \ \text{while}(0 \leq x < 1) \ {}\ 
\]
\[ \ell_2 : \ i := i - 1; \]
\[ \ell_3 : \ x := x + 1; \]
\}

Suppose we check:

\[ G \ (\neg \ell_0 \rightarrow 0 \leq i) \]

Using:

\[ E = \{0 \leq i\} \]

\[ p_0 \]
Example: Predicate Abstraction

\[\ell_0 : \ i := 1; \]
\[\ell_1 : \ \textbf{while}(0 \leq x < 1) \ {\}
\]
\[\ell_2 : \ i := i - 1; \]
\[\ell_3 : \ x := x + 1; \]

Suppose we check:

\[\mathcal{G} (\neg \ell_0 \rightarrow 0 \leq i)\]

Using:

\[E = \{0 \leq i\} \]

\[p_0\]
Example: Predicate Abstraction

\[ \ell_0 : \quad i := 1; \]
\[ \ell_1 : \quad \textbf{while}(0 \leq x < 1) \{ \]
\[ \quad \ell_2 : \quad i := i - 1; \]
\[ \quad \ell_3 : \quad x := x + 1; \]
\[ \} \]

Suppose we check:
\[ G (\neg \ell_0 \rightarrow 0 \leq i) \]

Using:
\[ E = \{0 \leq i\} \]

\[ p_0 \]
Example: Predicate Abstraction

\begin{align*}
\ell_0 &: \quad i := 1;
\ell_1 &: \quad \textbf{while}(0 \leq x < 1) \{ \\
\ell_2 &: \quad i := i - 1;
\ell_3 &: \quad x := x + 1;
\} \\
\end{align*}

Suppose we check:
\[ G (\neg \ell_0 \rightarrow 0 \leq i) \]

Using:
\[ E = \{0 \leq i\} \]

\[ p_0 \]
Example: Predicate Abstraction

\[ \ell_0 : \quad i := 1; \]
\[ \ell_1 : \quad \textbf{while}(0 \leq x < 1) \{ \]
\[ \ell_2 : \quad i := i - 1; \]
\[ \ell_3 : \quad x := x + 1; \]
\[ \} \]

Suppose we check:

\[ G \left( \neg \ell_0 \rightarrow 0 \leq i \right) \]

Using:

\[ E = \{ 0 \leq i \} \]
Example: Predicate Abstraction

\[ \ell_0 : \ i := 1; \]
\[ \ell_1 : \ \textbf{while}(0 \leq x < 1) \ { } \]
\[ \ell_2 : \ i := i - 1; \]
\[ \ell_3 : \ x := x + 1; \]

}\}


Does the property hold?

\[ \mathbf{G} \ (\neg \ell_0 \rightarrow 0 \leq i) \]
Example: Predicate Abstraction

\[ \ell_0 : \ i := 1; \]
\[ \ell_1 : \ \textbf{while}(0 \leq x < 1) \ {\}
\[ \ell_2 : \ i := i - 1; \]
\[ \ell_3 : \ x := x + 1; \]
\}

Does the property hold?

\[ \mathbf{G} (\neg \ell_0 \rightarrow 0 \leq i) \]

No.
Example: Predicate Abstraction

\[ \begin{align*}
\ell_0 & : \ i := 1; \\
\ell_1 & : \ \textbf{while}(0 \leq x < 1) \ { \{ \\
\ell_2 & : \ i := i - 1; \\
\ell_3 & : \ x := x + 1; \\
\} } 
\end{align*} \]

Does the property hold?

\[ \mathbf{G} (\neg \ell_0 \rightarrow 0 \leq i) \]

\textbf{No.} What’s a counterexample?
Example: Predicate Abstraction

\[
\ell_0 : \ i := 1;  \\
\ell_1 : \ \textbf{while}(0 \leq x < 1) \ \{  \\
\ell_2 : \ i := i - 1;  \\
\ell_3 : \ x := x + 1;  \\
\}  
\]

Does the property hold?

\[G(\neg \ell_0 \rightarrow 0 \leq i)\]

No. What’s a counterexample?
Example: Predicate Abstraction

\[ \ell_0: \ i := 1; \]
\[ \ell_1: \ \textbf{while}(0 \leq x < 1) \ \{ \]
\[ \ell_2: \ i := i - 1; \]
\[ \ell_3: \ x := x + 1; \]
\[ \} \]

Does the property hold?

\[ G(\neg \ell_0 \rightarrow 0 \leq i) \]

\textbf{No}. What’s a counterexample?

\[ (\ell_0, p_0) \]
Example: Predicate Abstraction

\[ \ell_0 : \quad i := 1; \]
\[ \ell_1 : \quad \textbf{while}(0 \leq x < 1) \{ \]
\[ \ell_2 : \quad i := i - 1; \]
\[ \ell_3 : \quad x := x + 1; \]
\[ \} \]

Does the property hold?
\[ \mathbf{G} (\neg \ell_0 \rightarrow 0 \leq i) \]

No. What’s a counterexample?

\[ (\ell_0, p_0) \]
\[ (\ell_1, p_0) \]
Example: Predicate Abstraction

\[
\ell_0 : \quad i := 1; \\
\ell_1 : \quad \textbf{while} (0 \leq x < 1) \{ \\
\quad \ell_2 : \quad i := i - 1; \\
\quad \ell_3 : \quad x := x + 1; \\
\} \\
\]

Does the property hold? 
\[ \mathbf{G} (\neg \ell_0 \rightarrow 0 \leq i) \]

\textbf{No.} What’s a counterexample?
\[(\ell_0, p_0)\]
\[(\ell_1, p_0)\]
\[(\ell_2, \neg p_0)\]
Spurious Counterexamples

\[ \ell_0 : \ i := 1; \]
\[ \ell_1 : \ \textbf{while}(0 \leq x < 1) \{ \]
\[ \ell_2 : \ i := i - 1; \]
\[ \ell_3 : \ x := x + 1; \]
\[ \} \]

Consider the KS path:

\[ (\ell_0, p_0) \]
\[ (\ell_1, p_0) \]
\[ (\ell_2, \neg p_0) \]

(recall that \( p_0 \Leftrightarrow 0 \leq i \))
\[\ell_0 : \quad i := 1;\]
\[\ell_1 : \quad \textbf{while}(0 \leq x < 1) \{\]
\[\ell_2 : \quad i := i - 1;\]
\[\ell_3 : \quad x := x + 1;\]
\[\}\]

Consider the KS path:
\[(\ell_0, p_0)\]
\[(\ell_1, p_0)\]
\[(\ell_2, \neg p_0)\]

(recall that \(p_0 \Leftrightarrow 0 \leq i\))

Consider the corresponding program path
Consider the KS path:

\[(\ell_0, p_0)\]
\[(\ell_1, p_0)\]
\[(\ell_2, \neg p_0)\]

(recall that \(p_0 \iff 0 \leq i\))

Consider the corresponding program path

\[
\ell_0 : \ i := 1;
\ell_1 : \ \textbf{while}(0 \leq x < 1) \ {\{ \}
\ell_2 : \ i := i - 1;
\ell_3 : \ x := x + 1;
\]

\[
\ell_0 : \ i := 1;
\]
Spurious Counterexamples

\[ \ell_0 : \ i := 1; \]
\[ \ell_1 : \ \text{while}(0 \leq x < 1) \ {\}\}
\[ \ell_2 : \ i := i - 1; \]
\[ \ell_3 : \ x := x + 1; \]

Consider the KS path:
\[(\ell_0, p_0)\]
\[(\ell_1, p_0)\]
\[(\ell_2, \neg p_0)\]

(recall that \(p_0 \Leftrightarrow 0 \leq i\))

Consider the corresponding program path

\[ \ell_0 : \ i := 1; \]
\[ \ell_1 : \ \text{assume}(0 \leq x < 1) \]
Consider the KS path:

\[(\ell_0, p_0)\]
\[(\ell_1, p_0)\]
\[(\ell_2, \neg p_0)\]

(recall that \(p_0 \iff 0 \leq i\))

Consider the corresponding program path

\[
\ell_0 : \quad i := 1; \\
\ell_1 : \quad \textbf{while}(0 \leq x < 1) \{ \\
\ell_2 : \quad i := i - 1; \\
\ell_3 : \quad x := x + 1; \\
\} \\
\{0 \leq i\}
\]
\[
\ell_0 : \quad i := 1; \\
\ell_1 : \quad \textbf{assume}(0 \leq x < 1)
\]
Consider the KS path:

\[(\ell_0, p_0)\]
\[(\ell_1, p_0)\]
\[(\ell_2, \neg p_0)\]

(recall that \( p_0 \Leftrightarrow 0 \leq i \))

Consider the corresponding program path

\( \ell_0 : \ i := 1; \)
\( \ell_1 : \ \textbf{while}(0 \leq x < 1) \{ \)
\( \ell_2 : \ i := i - 1; \)
\( \ell_3 : \ x := x + 1; \)
\}
Consider the KS path:

$$(\ell_0, p_0)$$
$$(\ell_1, p_0)$$
$$(\ell_2, \neg p_0)$$

(recall that $p_0 \iff 0 \leq i$)

Is this a valid Hoare triple?

Consider the corresponding program path

\begin{align*}
\ell_0 : & \quad i := 1; \\
\ell_1 : & \quad \textbf{while}(0 \leq x < 1) \{ \\
\ell_2 : & \quad i := i - 1; \\
\ell_3 : & \quad x := x + 1; \\
\} \\
\end{align*}

\begin{align*}
\ell_0 : & \quad i := 1; \\
\ell_1 : & \quad \textbf{assume}(0 \leq x < 1) \{i < 0\}
\end{align*}
Spurious Counterexamples

\[
\ell_0 : \quad i := 1; \\
\ell_1 : \quad \textbf{while}(0 \leq x < 1) \{ \\
\ell_2 : \quad i := i - 1; \\
\ell_3 : \quad x := x + 1; \\
\} \\
\{0 \leq i\}
\]

Consider the KS path:

\[
(\ell_0, p_0) \\
(\ell_1, p_0) \\
(\ell_2, \neg p_0)
\]

(recall that \(p_0 \iff 0 \leq i\))

Is this a valid Hoare triple?

1. \(\{0 \leq i\} \quad i := 1 \quad \{0 \leq i\}\)

Consider the corresponding program path

\[
\ell_0 : \quad i := 1; \\
\ell_1 : \quad \textbf{assume}(0 \leq x < 1) \\
\{i < 0\}
\]
Spurious Counterexamples

\[\ell_0 : \ i := 1;\]
\[\ell_1 : \ \text{while}(0 \leq x < 1) \{\]
\[\ell_2 : \ i := i - 1;\]
\[\ell_3 : \ x := x + 1;\]
\[
\]
Consider the KS path:

- \((\ell_0, p_0)\)
- \((\ell_1, p_0)\)
- \((\ell_2, \neg p_0)\)

(recall that \(p_0 \Leftrightarrow 0 \leq i\))

Is this a valid Hoare triple?

1. \(\{0 \leq i\} \ i := 1 \ \{0 \leq i\} \text{ Yes}\)

Consider the corresponding program path

\[\ell_0 : \ i := 1;\]
\[\ell_1 : \ \text{assume}(0 \leq x < 1) \]
\[\ell_0 : \ i := 1;\]
\[\ell_1 : \ \text{assume}(0 \leq x < 1) \]
\[
\]
Consider the KS path:

\[(\ell_0, p_0)\]
\[(\ell_1, p_0)\]
\[(\ell_2, \neg p_0)\]

(recall that \(p_0 \Leftrightarrow 0 \leq i\))

Is this a valid Hoare triple?

1. \(\{0 \leq i\} i := 1 \{0 \leq i\}\) Yes
2. \(\{0 \leq i\} \text{ assume}(0 \leq x < 1) \{0 > i\}\)

Consider the corresponding program path

\(\ell_0 : \quad i := 1;\)
\(\ell_1 : \quad \textbf{while}(0 \leq x < 1) \{\)
\(\ell_2 : \quad i := i - 1;\)
\(\ell_3 : \quad x := x + 1;\)
\}
Consider the KS path:

\[
(\ell_0, p_0) \\
(\ell_1, p_0) \\
(\ell_2, \neg p_0)
\]

(recall that \( p_0 \Leftrightarrow 0 \leq i \))

Is this a valid Hoare triple?

1. \( \{0 \leq i\} \; i := 1 \; \{0 \leq i\} \) \textbf{Yes}

2. \( \{0 \leq i\} \; \textbf{assume}(0 \leq x < 1) \; \{0 > i\} \) \textbf{No}

Consider the corresponding program path:

\( \ell_0 : \ i := 1; \)
\( \ell_1 : \ \textbf{while}(0 \leq x < 1) \{ \)
\( \ell_2 : \ i := i - 1; \)
\( \ell_3 : \ x := x + 1; \)
\( \} \)

\( \ell_0 : \ i := 1; \)
\( \ell_1 : \ \textbf{assume}(0 \leq x < 1) \)
\( \{i < 0\} \)
Consider the KS path:

\[(\ell_0, p_0) \quad (\ell_1, p_0) \quad (\ell_2, \neg p_0)\]

(recall that \( p_0 \iff 0 \leq i \))

Is this a valid Hoare triple?

1. \( \{0 \leq i\} \ i := 1 \ \{0 \leq i\} \quad \text{Yes} \\
2. \( \{0 \leq i\} \ \text{assume}(0 \leq x < 1) \ \{0 > i\} \quad \text{No} \\

Consider the corresponding program path

This is how we know that the counterexample is spurious
Abstraction Refinement

We want to make the abstraction more precise: add more predicates
We want to make the abstraction more precise: add more predicates

▶ At the very least, eliminate this counterexample
We want to make the abstraction more precise: add more predicates

▶ At the very least, eliminate this counterexample

▶ Hopefully, many more brought about by same “cause”
We want to make the abstraction more precise: add more predicates
- At the very least, eliminate this counterexample
- Hopefully, many more brought about by same “cause”

Called **counterexample-guided abstraction refinement** (CEGAR)
We want to make the abstraction more precise: add more predicates
  ▶ At the very least, eliminate this counterexample
  ▶ Hopefully, many more brought about by same “cause”

Called **counterexample-guided abstraction refinement** (CEGAR)
  ▶ E. Clarke, O. Grumberg, S. Jha, Y. Lu, H. Veith, 2000
We want to make the abstraction more precise: add more predicates

- At the very least, eliminate this counterexample
- Hopefully, many more brought about by same “cause”

Called **counterexample-guided abstraction refinement** (CEGAR)


Main technique behind all modern software model checking
We want to make the abstraction more precise: add more predicates
  ▶ At the very least, eliminate this counterexample
  ▶ Hopefully, many more brought about by same “cause”

Called **counterexample-guided abstraction refinement** (CEGAR)
  ▶ E. Clarke, O. Grumberg, S. Jha, Y. Lu, H. Veith, 2000

Main technique behind all modern software model checking
  1. Start with a simple, automatic abstraction
We want to make the abstraction more precise: add more predicates

- At the very least, eliminate this counterexample
- Hopefully, many more brought about by same “cause”

Called **counterexample-guided abstraction refinement** (CEGAR)


Main technique behind all modern software model checking

1. Start with a simple, automatic abstraction
2. Search for counterexamples
Abstraction Refinement

We want to make the abstraction more precise: add more predicates
  ▶ At the very least, eliminate this counterexample
  ▶ Hopefully, many more brought about by same “cause”

Called **counterexample-guided abstraction refinement** (CEGAR)
  ▶ E. Clarke, O. Grumberg, S. Jha, Y. Lu, H. Veith, 2000

Main technique behind all modern software model checking
  1. Start with a simple, automatic abstraction
  2. Search for counterexamples
  3. Refine spurious counterexamples, building model on-demand
We want to make the abstraction more precise: add more predicates
  ▶ At the very least, eliminate this counterexample
  ▶ Hopefully, many more brought about by same “cause”

Called **counterexample-guided abstraction refinement** (CEGAR)
  ▶ E. Clarke, O. Grumberg, S. Jha, Y. Lu, H. Veith, 2000

Main technique behind all modern software model checking
  1. Start with a simple, automatic abstraction
  2. Search for counterexamples
  3. Refine spurious counterexamples, building model on-demand
  4. Continue until real counterexample, or property holds
\[ \ell_0 : \quad i := 1; \]
\[ \ell_1 : \quad \textbf{while}(0 \leq x < 1) \{ \]
\[ \ell_2 : \quad i := i - 1; \]
\[ \ell_3 : \quad x := x + 1; \]
\[ \} \]

What caused this?

\[ (\ell_0, p_0) (\ell_1, p_0) (\ell_2, \neg p_0) \]
\[ \ell_0 : \quad i := 1; \]
\[ \ell_1 : \quad \textbf{while}(0 \leq x < 1) \{ \]
\[ \ell_2 : \quad i := i - 1; \]
\[ \ell_3 : \quad x := x + 1; \]
\[ \}\]

What caused this?

\((\ell_0, p_0) (\ell_1, p_0) (\ell_2, \neg p_0)\)
\( \ell_0 : \ i := 1; \)
\( \ell_1 : \ \textbf{while}(0 \leq x < 1) \ { \}
\( \ell_2 : \ i := i - 1; \)
\( \ell_3 : \ x := x + 1; \)

What caused this?

\( (\ell_0, p_0) (\ell_1, p_0) (\ell_2, \neg p_0) \)

We had \( \neg \text{Pred}(0 \leq x < 1, \{p_0\}) = \text{true} \)
\[ \ell_0 : \quad i := 1; \]
\[ \ell_1 : \quad \text{while}(0 \leq x < 1) \{ \]
\[ \ell_2 : \quad i := i - 1; \]
\[ \ell_3 : \quad x := x + 1; \]
\[ \} \]

What caused this?

\((\ell_0, p_0) (\ell_1, p_0) (\ell_2, \neg p_0)\)

We had \(\neg \text{Pred}(0 \leq x < 1, \{p_0\}) = \text{true}\)

...and \(\neg p_0 \Rightarrow \text{true}\)
\[ \ell_0 : \ i := 1; \]
\[ \ell_1 : \ \text{while}(0 \leq x < 1) \ { \}
\[ \ell_2 : \ i := i - 1; \]
\[ \ell_3 : \ x := x + 1; \]
\]

What caused this?

\[(\ell_0, p_0) (\ell_1, p_0) (\ell_2, \neg p_0)\]

We had \(\neg \text{Pred}(0 \leq x < 1, \{p_0\}) = true\)

...and \(\neg p_0 \Rightarrow true\)

How do we fix it?
\[ \ell_0 : \quad i := 1; \]
\[ \ell_1 : \quad \textbf{while}(0 \leq x < 1) \{ \]
\[ \ell_2 : \quad i := i - 1; \]
\[ \ell_3 : \quad x := x + 1; \]
\[ \} \]

What caused this?
\[(\ell_0, p_0) (\ell_1, p_0) (\ell_2, \neg p_0)\]

We had \(\neg \text{Pred}(0 \leq x < 1, \{p_0\}) = \text{true}\)

...and \(\neg p_0 \Rightarrow \text{true}\)

How do we fix it?
\[ E = \{0 \leq i, 0 \leq x < 1\} \]
\[ p_0 \quad p_1 \]
Example: Abstraction Refinement

\[ \ell_0 : i := 1; \]
\[ \ell_1 : \textbf{while}(0 \leq x < 1) \{ \]
\[ \ell_2 : i := i - 1; \]
\[ \ell_3 : x := x + 1; \]
\[ \} \]

\[ E = \{ 0 \leq i, 0 \leq x < 1 \} \]

\[ p_0 \]
\[ p_1 \]
Example: Abstraction Refinement

\[ \ell_0 : i := 1; \]
\[ \ell_1 : \text{while} (0 \leq x < 1) \{ \]
\[ \ell_2 : i := i - 1; \]
\[ \ell_3 : x := x + 1; \]
\[ \} \]

\[ E = \{0 \leq i, 0 \leq x < 1\} \]

\[ p_0, \overline{p}_1 \]

\[ \ell_0 \]

\[ \ell_1 \]

\[ \ell_2 \]

\[ \ell_3 \]

\[ \text{skip} \]

\[ \overline{p}_0, p_1 \]
Example: Abstraction Refinement

\[ \ell_0 : i := 1; \]
\[ \ell_1 : \text{while}(0 \leq x < 1) \{ \]
\[ \ell_2 : i := i - 1; \]
\[ \ell_3 : x := x + 1; \]
\[ \} \]
\[ \ell_e : \text{skip} \]

\[ E = \{0 \leq i, 0 \leq x < 1\} \]

\[ \begin{align*}
\ell_0 & : p_0, \overline{p_1} \\
\ell_1 & : p_0, \overline{p_1} \\
\ell_2 & : p_0, p_1 \\
\ell_3 & : \overline{p_0}, p_1 \\
\ell_e & : p_0, \overline{p_1}
\end{align*} \]
Example: Abstraction Refinement

\[ \ell_0 : i := 1; \]
\[ \ell_1 : \text{while}(0 \leq x < 1) \{ \]
\[ \ell_2 : i := i - 1; \]
\[ \ell_3 : x := x + 1; \]
\[ \} \]
\[ \ell_e : \text{skip} \]

\[ E = \{0 \leq i, 0 \leq x < 1\} \]

\[ p_0, \overline{p}_1 \]

\[ p_0, p_1 \]

\[ \overline{p}_0, \overline{p}_1 \]

\[ \overline{p}_0, p_1 \]

\[ \overline{p}_1, \overline{p}_1 \]

\[ \overline{p}_1, p_1 \]

\[ \text{skip} \]

\[ \text{skip} \]

\[ \text{skip} \]

\[ \text{skip} \]

\[ \text{skip} \]

\[ \text{skip} \]

\[ \text{skip} \]
Example: Abstraction Refinement

\[ \ell_0 : i := 1; \]
\[ \ell_1 : \text{while}(0 \leq x < 1) \{ \]
\[ \ell_2 : i := i - 1; \]
\[ \ell_3 : x := x + 1; \]
\[ \} \]
\[ \ell_e : \text{skip} \]

\[ E = \{ 0 \leq i, 0 \leq x < 1 \} \]
Example: Abstraction Refinement

\[ \ell_0 : i := 1; \]
\[ \ell_1 : \textbf{while}(0 \leq x < 1) \{ \]
\[ \ell_2 : i := i - 1; \]
\[ \ell_3 : x := x + 1; \]
\[ \} \]
\[ \ell_e : \textbf{skip} \]

\[ E = \{0 \leq i, 0 \leq x < 1\} \]

Is there a counterexample?
Example: Abstraction Refinement

\[ \ell_0 : i := 1; \]
\[ \ell_1 : \text{while}(0 \leq x < 1) \{ \]
\[ \ell_2 : i := i - 1; \]
\[ \ell_3 : x := x + 1; \]
\[ \} \]
\[ \ell_e : \text{skip} \]

\[ E = \{0 \leq i, 0 \leq x < 1 \} \]

\[ p_0 \]

\[ p_1 \]
Example: Abstraction Refinement

\[
\ell_0 : i := 1;
\ell_1 : \text{while}(0 \leq x < 1) \{
\ell_2 : i := i - 1;
\ell_3 : x := x + 1;
\}
\ell_e : \text{skip}
\]

\[E = \{0 \leq i, 0 \leq x < 1\}\]

Is this valid?
\[
\{0 \leq i \land 0 \leq x < 1\}
\]
\[
i := 1;
\text{assume}(0 \leq x < 1)
\]
\[
i := i - 1;
\{i < 0 \land 0 \leq x < 1\}\]
Lazy Abstraction

After the second counterexample, it seems $x = 1$ is relevant
After the second counterexample, it seems $x = 1$ is relevant.

We should really add this to our abstraction set $E$. 
Lazy Abstraction

After the second counterexample, it seems $x = 1$ is relevant.

We should really add this to our abstraction set $E$.

This is turning into a lot of work!
Lazy Abstraction

After the second counterexample, it seems \( x = 1 \) is relevant

We should really add this to our abstraction set \( E \)

This is turning into a lot of work!
  ▶ Now we have 8 initial states...
After the second counterexample, it seems $x = 1$ is relevant

We should really add this to our abstraction set $E$

This is turning into a lot of work!
  - Now we have 8 initial states...
  - $\#\text{loc} \times 2^{|E|}$ states in general
Lazy Abstraction

After the second counterexample, it seems $x = 1$ is relevant

We should really add this to our abstraction set $E$

This is turning into a lot of work!

- Now we have 8 initial states...
- $\#loc \times 2^{|E|}$ states in general
- There must be a better way!
Lazy Abstraction

After the second counterexample, it seems $x = 1$ is relevant.

We should really add this to our abstraction set $E$.

This is turning into a lot of work!

- Now we have 8 initial states...
- \#loc $\times 2^{|E|}$ states in general
- There must be a better way!

**Idea:** Don’t refine error-free parts of the abstraction.
Example: Lazy Abstraction

\[
\ell_0 : i := 1; \\
\ell_1 : \text{while}(0 \leq x < 1) \{ \\
\ell_2 : i := i - 1; \\
\ell_3 : x := x + 1; \\
\} \\
\ell_e : \text{skip}
\]

\[E = \{0 \leq i, 0 \leq x < 1\}\]

\[p_0, p_1\]
Example: Lazy Abstraction

ℓ₀ : i := 1;
ℓ₁ : while (0 ≤ x < 1) {
   ℓ₂ : i := i - 1;
   ℓ₃ : x := x + 1;
}
ℓₑ : skip

E = \{0 ≤ i, 0 ≤ x < 1\}

Don’t need to update left side with

p₂ ⇔ i = 1
Example: Lazy Abstraction

\[
\ell_0: i := 1; \\
\ell_1: \textbf{while}(0 \leq x < 1) \{ \\
\ell_2: \ i := i - 1; \\
\ell_3: \ x := x + 1; \\
\} \\
\ell_e: \textbf{skip}
\]

\[E = \{0 \leq i, 0 \leq x < 1\}\]

Don’t need to update left side with \(p_2 \iff i = 1\)
Example: Lazy Abstraction

\[ \ell_0 : i := 1; \]
\[ \ell_1 : \textbf{while}(0 \leq x < 1) \{ \]
\[ \ell_2 : i := i - 1; \]
\[ \ell_3 : x := x + 1; \]
\[ \} \]
\[ \ell_e : \textbf{skip} \]

\[ E = \{0 \leq i, 0 \leq x < 1\} \]

Don’t need to update left side with

\[ p_2 \iff i = 1 \]
Example: Lazy Abstraction

\[
\ell_0 : i := 1; \\
\ell_1 : \textbf{while}(0 \leq x < 1) \{ \\
\ell_2 : i := i - 1; \\
\ell_3 : x := x + 1; \\
\} \\
\ell_e : \textbf{skip}
\]

\[
E = \{0 \leq i, 0 \leq x < 1\}
\]

Don’t need to update left side with \(p_2 \iff i = 1\)

Now there’s no counterexample
Lazy abstraction was developed by Henzinger et al, 2002
Lazy abstraction was developed by Henzinger et al, 2002

Combines on-demand search with “refinement where necessary”
Lazy abstraction was developed by Henzinger et al, 2002

Combines on-demand search with “refinement where necessary”

Key data structure: reachability tree
More on Lazy Abstraction

Lazy abstraction was developed by Henzinger et al, 2002

Combines on-demand search with “refinement where necessary”

Key data structure: reachability tree

1. Pick an abstract initial state
More on Lazy Abstraction

Lazy abstraction was developed by Henzinger et al., 2002

Combines on-demand search with “refinement where necessary”

Key data structure: reachability tree

1. Pick an abstract initial state
2. Add children by computing abstract transitions
More on Lazy Abstraction

Lazy abstraction was developed by Henzinger et al, 2002

Combines on-demand search with “refinement where necessary”

Key data structure: reachability tree

1. Pick an abstract initial state
2. Add children by computing abstract transitions
3. Only refine subtrees that could contain errors
Lazy abstraction was developed by Henzinger et al, 2002

Combines on-demand search with "refinement where necessary"

Key data structure: reachability tree
   1. Pick an abstract initial state
   2. Add children by computing abstract transitions
   3. Only refine subtrees that could contain errors

In practice, this approach gives drastic performance improvements
\[ \ell_0 : i := 1; \]
\[ \ell_1 : \text{while}(0 \leq x < 1) \{ \]
\[ \ell_2 : i := i - 1; \]
\[ \ell_3 : x := x + 1; \]
\[ \} \]
\[ \ell_e : \text{skip} \]

\[ E = \{0 \leq i, 0 \leq x < 1, i = 1\} \]

\[ \begin{align*}
\ell_0 & \quad p_0, p_1 \\
\ell_1 & \quad p_0, p_1 \\
\ell_2 & \quad p_0, p_1, p_2 \\
\ell_3 & \quad p_0, p_1, p_2 \\
\ell_e & \quad p_0, p_1 \\
\end{align*} \]
\( \{ \text{true} \} \)

\[ \ell_0 : i := 1; \]

\[ \ell_1 : \text{while} (0 \leq x < 1) \{ \]

\[ \ell_2 : i := i - 1; \]

\[ \ell_3 : x := x + 1; \]

\} \]

\[ \ell_e : \text{skip} \]

\[ E = \{ 0 \leq i, 0 \leq x < 1, i = 1 \} \]

\[ p_0 \]

\[ p_1 \]

\[ p_2 \]
\[
\{ \text{true} \} \\
\ell_0 : i := 1; \\
\{ 0 \leq i \land i = 1 \} \\
\ell_1 : \textbf{while}(0 \leq x < 1) \{ \\
\ell_2 : i := i - 1; \\
\ell_3 : x := x + 1; \\
\} \\
\ell_e : \textbf{skip} \\
E = \{ 0 \leq i, 0 \leq x < 1, i = 1 \} \\
\]

\begin{align*}
&\ell_0 \\
&\ell_1 \\
&\ell_2 \\
&\ell_3 \\
&\ell_e
\end{align*}
Proofs from Abstractions

\{true\}
\ell_0 : i := 1;
\{0 \leq i \land i = 1\}
\ell_1 : \textbf{while}(0 \leq x < 1) \{ \\
\{0 \leq i \land 0 \leq x < 1 \land i = 1\}
\ell_2 : i := i - 1;
\{0 \leq i \land 0 \leq x < 1\}
\ell_3 : x := x + 1;
\}
\ell_e : \textbf{skip}

E = \{0 \leq i, 0 \leq x < 1, i = 1\}
$p_0$, $p_1$, $p_2$
Proofs from Abstractions

\{true\}
\ell_0 : i := 1;
\{0 \leq i \land i = 1\}
\ell_1 : \textbf{while}(0 \leq x < 1) \{ \\
\{0 \leq i \land 0 \leq x < 1 \land i = 1\}
\ell_2 : i := i - 1; \\
\{0 \leq i \land 0 \leq x < 1\}
\ell_3 : x := x + 1; \\
\{0 \leq i \land \neg(0 \leq x < 1)\}
\ell_e : \textbf{skip}

E = \{0 \leq i, 0 \leq x < 1, i = 1\}

\begin{align*}
p_0 & \quad p_1 & \quad p_2 \\
\ell_0 & \quad \ell_0 & \quad \ell_0 & \quad \ell_0 \\
\ell_1 & \quad \ell_1 & \quad \ell_1 & \\
\ell_e & \quad \ell_2 & \quad \ell_3 &
\end{align*}
Proofs from Abstractions

\[
\{ \text{true} \} \\
\ell_0 : i := 1; \\
\quad \{ 0 \leq i \land i = 1 \} \\
\ell_1 : \textbf{while}(0 \leq x < 1) \{ \\
\quad \{ 0 \leq i \land 0 \leq x < 1 \land i = 1 \} \\
\ell_2 : i := i - 1; \\
\quad \{ 0 \leq i \land 0 \leq x < 1 \land i \neq 1 \} \\
\ell_3 : x := x + 1; \\
\quad \{ 0 \leq i \land \neg(0 \leq x < 1) \land i \neq 1 \} \\
\} \\
\ell_e : \textbf{skip}
\]

\[
E = \{ 0 \leq i, \begin{array}{c} 0 \leq x < 1, \end{array} i = 1 \} \\
\begin{array}{c}
p_0 \quad p_1 \quad p_2 \end{array}
\]
Proofs from Abstractions

\{\textit{true}\}\]
\ell_0 : \text{\textbf{i} := 1;}
\quad \{0 \leq i \land i = 1\}
\ell_1 : \textbf{while}(0 \leq x < 1) \{ \\
\quad \{0 \leq i \land 0 \leq x < 1 \land i = 1\}
\ell_2 : \text{\textbf{i} := i - 1;}
\quad \{0 \leq i \land 0 \leq x < 1 \land i \neq 1\}
\ell_3 : \text{\textbf{x} := x + 1;}
\quad \{0 \leq i \land \neg(0 \leq x < 1) \land i \neq 1\}
\}
\ell_e : \textbf{skip}

\[E = \{0 \leq i, 0 \leq x < 1, i = 1\}\]

These annotations are sufficient to prove the property

\[E = \{0 \leq i, 0 \leq x < 1, i = 1\}\]
Proofs from Abstractions

These annotations are sufficient to prove the property

Suppose we wanted to verify

\{true\} Prog \{0 \leq i\}

{\text{true}\
\ell_0 : i := 1;
\{0 \leq i \land i = 1\}
\ell_1 : \text{while}(0 \leq x < 1) \{\
\{0 \leq i \land 0 \leq x < 1 \land i = 1\}
\ell_2 : i := i - 1;
\{0 \leq i \land 0 \leq x < 1 \land i \neq 1\}
\ell_3 : x := x + 1;
\{0 \leq i \land \neg(0 \leq x < 1) \land i \neq 1\}\}
\ell_e : \text{skip}

\[ E = \{0 \leq i, 0 \leq x < 1, i = 1\} \]
These annotations are sufficient to prove the property

Suppose we wanted to verify

\[ \{ \text{true} \} \text{Prog} \{ 0 \leq i \} \]

What is our loop invariant?
Proofs from Abstractions

{true}
\ell_0 : i := 1;
\{0 \leq i \land i = 1\}
\ell_1 : \textbf{while}(0 \leq x < 1) \{
\{0 \leq i \land 0 \leq x < 1 \land i = 1\}
\ell_2 : i := i - 1;
\{0 \leq i \land 0 \leq x < 1 \land i \neq 1\}
\ell_3 : x := x + 1;
\{0 \leq i \land \neg(0 \leq x < 1) \land i \neq 1\}
\}
\ell_e : \textbf{skip}

These annotations are sufficient to prove the property

Suppose we wanted to verify
\{true\} Prog \{0 \leq i\}

What is our loop invariant?

\begin{align*}
(0 \leq i \land i = 1) \\
\lor (0 \leq i \land 0 \leq x < 1 \land i = 1) \\
\lor (0 \leq i \land 0 \leq x < 1 \land i \neq 1) \\
\lor (0 \leq i \land \neg(0 \leq x < 1) \land i \neq 1)
\end{align*}

E = \{0 \leq i, 0 \leq x < 1, i = 1\}
\begin{array}{c}
p_0 \\
p_1 \\
p_2
\end{array}
\[
\{ \text{true} \} \\
\ell_0 : i := 1; \\
\{ 0 \leq i \land i = 1 \} \\
\ell_1 : \text{while}(0 \leq x < 1) \{ \\
\{ 0 \leq i \land 0 \leq x < 1 \land i = 1 \} \\
\ell_2 : i := i - 1; \\
\{ 0 \leq i \land 0 \leq x < 1 \land i \neq 1 \} \\
\ell_3 : x := x + 1; \\
\{ 0 \leq i \land \neg(0 \leq x < 1) \land i \neq 1 \} \\
\ell_e : \text{skip} \\
\]

These annotations are sufficient to prove the property

Suppose we wanted to verify

\[
\{ \text{true} \} \text{ Prog } \{ 0 \leq i \}
\]

What is our loop invariant?

\[
(0 \leq i \land i = 1) \\
\lor (0 \leq i \land 0 \leq x < 1 \land i = 1) \\
\lor (0 \leq i \land 0 \leq x < 1 \land i \neq 1) \\
\lor (0 \leq i \land \neg(0 \leq x < 1) \land i \neq 1) \\
\Leftrightarrow 0 \leq i
\]
These annotations are sufficient to prove the property

Suppose we wanted to verify

\[ \text{true} \] \text{Prog} \{0 \leq i\}

What is our loop invariant?

\[ (0 \leq i \land i = 1) \]
\[ \lor (0 \leq i \land 0 \leq x < 1 \land i = 1) \]
\[ \lor (0 \leq i \land 0 \leq x < 1 \land i \neq 1) \]
\[ \leftrightarrow \]
\[ 0 \leq i \]

CEGAR automatically constructs deductive proofs!
Suppose we wanted to verify:

\[
\begin{align*}
\{true\} \\
\ell_0 &: i := 10; \\
\ell_1 &: \textbf{while}(0 \leq x < 10) \{ \\
\ell_2 &: \ i := i - 1; \\
\ell_3 &: \ x := x + 1; \\
& \} \\
\ell_e &: \textbf{skip} \\
& \{0 \leq i\}
\end{align*}
\]
Suppose we wanted to verify:

\[
\begin{align*}
\{true\} \\
\ell_0 &: i := 10; \\
\ell_1 &: \text{while}(0 \leq x < 10) \{ \\
\ell_2 &: i := i - 1; \\
\ell_3 &: x := x + 1; \\
\} \\
\ell_e &: \text{skip} \\
\{0 \leq i\}
\end{align*}
\]

How would we do it by hand?

Finding the right predicates early is crucial.
Suppose we wanted to verify:

\[
\begin{align*}
\{true\} \\
\ell_0 &: i := 10; \\
\ell_1 &: \textbf{while}\,(0 \leq x < 10) \{ \\
\ell_2 &: i := i - 1; \\
\ell_3 &: x := x + 1; \\
\} \\
\ell_e &: \textbf{skip} \\
\{0 \leq i\}
\end{align*}
\]

How would we do it by hand?
- Find the invariant \(0 \leq i - x\)

How would CEGAR do it?

Finding the right predicates early is crucial.
Suppose we wanted to verify:

\[
\{\text{true}\}
\]

\[
\ell_0 : i := 10;
\]

\[
\ell_1 : \textbf{while} (0 \leq x < 10) \{ \\
\ell_2 : i := i - 1; \\
\ell_3 : x := x + 1; \\
\}
\]

\[
\ell_e : \textbf{skip} \\
\{ 0 \leq i \}
\]

How would we do it by hand?

- Find the invariant \(0 \leq i - x\)

How would CEGAR do it?

- Find \(i = 10, x = 9\)
Suppose we wanted to verify:

\[
\begin{align*}
\{true\} \\
\ell_0 : & i := 10; \\
\ell_1 : while(0 \leq x < 10) \{ \\
\ell_2 : & i := i - 1; \\
\ell_3 : & x := x + 1; \\
\} \\
\ell_e : skip \\
\{0 \leq i\}
\end{align*}
\]

How would we do it by hand?
- Find the invariant \(0 \leq i - x\)

How would CEGAR do it?
- Find \(i = 10, x = 9\)
- Find \(i = 10, x = 8\)
Suppose we wanted to verify:

\[
\begin{align*}
\{ & true \} \\
\ell_0 & : i := 10; \\
\ell_1 & : \textbf{while} (0 \leq x < 10) \{ \\
\ell_2 & : i := i - 1; \\
\ell_3 & : x := x + 1; \\
\} \\
\ell_e & : \textbf{skip} \\
\{ & 0 \leq i \}
\end{align*}
\]

How would we do it by hand?

- Find the invariant \( 0 \leq i - x \)

How would CEGAR do it?

- Find \( i = 10, x = 9 \)
- Find \( i = 10, x = 8 \)
- \( \ldots \)
Suppose we wanted to verify:

\[
\begin{align*}
\{ \text{true} \} \\
\ell_0 & : i := 10; \\
\ell_1 & : \textbf{while}(0 \leq x < 10) \{ \\
\ell_2 & : i := i - 1; \\
\ell_3 & : x := x + 1; \\
\} \\
\ell_e & : \textbf{skip} \\
\{ 0 \leq i \}
\end{align*}
\]

How would we do it by hand?
- Find the invariant \( 0 \leq i - x \)

How would CEGAR do it?
- Find \( i = 10, x = 9 \)
- Find \( i = 10, x = 8 \)
- ... 
- Find \( i = 9 \)
Suppose we wanted to verify:

\[
\{\text{true}\} \\
\ell_0 : i := 10; \\
\ell_1 : \textbf{while}(0 \leq x < 10) \{ \\
\ell_2 : i := i - 1; \\
\ell_3 : x := x + 1; \\
\} \\
\ell_e : \textbf{skip} \\
\{0 \leq i\}
\]

How would we do it by hand?
- Find the invariant $0 \leq i - x$

How would CEGAR do it?
- Find $i = 10, x = 9$
- Find $i = 10, x = 8$
- ...$
- Find $i = 9$
- ...
Suppose we wanted to verify:

\[
\{\text{true}\} \\
\ell_0 : i := 10; \\
\ell_1 : \textbf{while} (0 \leq x < 10) \{ \\
\ell_2 : i := i - 1; \\
\ell_3 : x := x + 1; \\
\} \\
\ell_e : \textbf{skip} \\
\{0 \leq i\}
\]

How would we do it by hand?

- Find the invariant \(0 \leq i - x\)

How would CEGAR do it?

- Find \(i = 10, x = 9\)
- Find \(i = 10, x = 8\)
- ...  
- Find \(i = 9\)  
- ...  

Finding the right predicates early is crucial.
Learning Predicates

Before, we found new predicates by intuition
Before, we found new predicates by intuition

Model checkers must do it automatically
Before, we found new predicates by intuition

Model checkers must do it automatically

**Key tool:** SMT solver
Learning Predicates

Before, we found new predicates by intuition

Model checkers must do it automatically

**Key tool:** SMT solver

- Given counterexample \( (\ell_1, \phi_1), \ldots, (\ell_n, \phi_n) \) generate \( \phi_{\text{path}} \)
Before, we found new predicates by intuition

Model checkers must do it automatically

**Key tool:** SMT solver

- Given counterexample \((\ell_1, \phi_1), \ldots, (\ell_n, \phi_n)\) generate \(\phi_{\text{path}}\)
- \(\phi_{\text{path}}\) is sat iff \((\ell_1, \phi_1), \ldots, (\ell_n, \phi_n)\) not spurious
Learning Predicates

Before, we found new predicates by intuition

Model checkers must do it automatically

**Key tool**: SMT solver

- Given counterexample $(\ell_1, \phi_1), \ldots, (\ell_n, \phi_n)$ generate $\phi_{\text{path}}$
- $\phi_{\text{path}}$ is sat iff $(\ell_1, \phi_1), \ldots, (\ell_n, \phi_n)$ not spurious
- If $\phi_{\text{path}}$ unsat, extract predicates from “witness”
Before, we found new predicates by intuition

Model checkers must do it automatically

**Key tool:** SMT solver

- Given counterexample $(\ell_1, \phi_1), \ldots, (\ell_n, \phi_n)$ generate $\phi_{\text{path}}$
- $\phi_{\text{path}}$ is sat iff $(\ell_1, \phi_1), \ldots, (\ell_n, \phi_n)$ not spurious
- If $\phi_{\text{path}}$ unsat, extract predicates from “witness”

Intuitively,
Learning Predicates

Before, we found new predicates by intuition

Model checkers must do it automatically

**Key tool**: SMT solver

- Given counterexample \((\ell_1, \phi_1), \ldots, (\ell_n, \phi_n)\) generate \(\phi_{\text{path}}\)
- \(\phi_{\text{path}}\) is sat iff \((\ell_1, \phi_1), \ldots, (\ell_n, \phi_n)\) not spurious
- If \(\phi_{\text{path}}\) unsat, extract predicates from “witness”

Intuitively,

- \(\phi_{\text{path}}\) simulates executing the counterexample path
Learning Predicates

Before, we found new predicates by intuition

Model checkers must do it automatically

**Key tool**: SMT solver

- Given counterexample \((\ell_1, \phi_1), \ldots, (\ell_n, \phi_n)\) generate \(\phi_{\text{path}}\)
- \(\phi_{\text{path}}\) is sat iff \((\ell_1, \phi_1), \ldots, (\ell_n, \phi_n)\) not spurious
- If \(\phi_{\text{path}}\) unsat, extract predicates from “witness”

Intuitively,

- \(\phi_{\text{path}}\) simulates executing the counterexample path
- If execution completes without error, path is valid counterexample
Learning Predicates

Before, we found new predicates by intuition

Model checkers must do it automatically

**Key tool:** SMT solver

- Given counterexample \((\ell_1, \phi_1), \ldots, (\ell_n, \phi_n)\) generate \(\phi_{\text{path}}\)
- \(\phi_{\text{path}}\) is sat iff \((\ell_1, \phi_1), \ldots, (\ell_n, \phi_n)\) not spurious
- If \(\phi_{\text{path}}\) unsat, extract predicates from “witness”

Intuitively,

- \(\phi_{\text{path}}\) simulates executing the counterexample path
- If execution completes without error, path is valid counterexample
- Otherwise, take an observation that explains why the path won’t execute
To build $\phi_{\text{path}}$, we’ll put path in static single-assignment (SSA) form.
To build $\phi_{\text{path}}$, we’ll put path in **static single-assignment** (SSA) form

Assume we’re given a path with only assign, **assume**, **assert**
To build $\phi_{\text{path}}$, we’ll put path in \textit{static single-assignment} (SSA) form

Assume we’re given a path with only assign, \textit{assume}, \textit{assert}

Each variable is only assigned once:
To build $\phi_{\text{path}}$, we’ll put path in static single-assignment (SSA) form.

Assume we’re given a path with only assign, assume, assert.

Each variable is only assigned once:

1. Attach subscripts to vars, starting at 0.
To build $\phi_{\text{path}}$, we’ll put path in **static single-assignment** (SSA) form.

Assume we’re given a path with only assign, **assume**, **assert**

Each variable is only assigned once:
1. Attach subscripts to vars, starting at 0
2. Each time a variable is assigned, increment its subscript
To build $\phi_{\text{path}}$, we’ll put path in static single-assignment (SSA) form

Assume we’re given a path with only assign, assume, assert

Each variable is only assigned once:
1. Attach subscripts to vars, starting at 0
2. Each time a variable is assigned, increment its subscript
3. All reads of the variable use the must recent subscript
Building Path Formulas

We’re given a path \((\ell_1, \phi_1), \ldots, (\ell_n, \phi_n)\)
We’re given a path \((\ell_1, \phi_1), \ldots, (\ell_n, \phi_n)\)

1. Build an annotated path by including \(\phi_1, \ldots, \phi_n\) as assertions
Building Path Formulas

We’re given a path \((\ell_1, \phi_1), \ldots, (\ell_n, \phi_n)\)

1. Build an annotated path by including \(\phi_1, \ldots, \phi_n\) as assertions

\[
\begin{align*}
\text{assert } & 0 \leq i \\
i & := 1 \\
\text{assert } & 0 \leq i \\
\text{assume } & 0 \leq x < 1 \\
\text{assert } & \neg(0 \leq i) \\
i & := i - 1
\end{align*}
\]
Building Path Formulas

We’re given a path \((\ell_1, \phi_1), \ldots, (\ell_n, \phi_n)\)

1. Build an annotated path by including \(\phi_1, \ldots, \phi_n\) as assertions
2. Convert the path into SSA form

\begin{align*}
\text{assert } 0 &\leq i \\
i &:= 1 \\
\text{assert } 0 &\leq i \\
\text{assume } 0 &\leq x < 1 \\
\text{assert } \neg(0 &\leq i) \\
i &:= i - 1
\end{align*}
Building Path Formulas

We’re given a path $(\ell_1, \phi_1), \ldots, (\ell_n, \phi_n)$

1. Build an annotated path by including $\phi_1, \ldots, \phi_n$ as assertions
2. Convert the path into SSA form

```
assert 0 \leq i
i := 1
assert 0 \leq i
assume 0 \leq x < 1
assert \neg(0 \leq i)
i := i - 1
assert 0 \leq i_0
i_1 := 1
assert 0 \leq i_1
assume 0 \leq x_0 < 1
assert \neg(0 \leq i_1)
i_0 := i_1 - 1
```
We’re given a path \((\ell_1, \phi_1), \ldots, (\ell_n, \phi_n)\)

1. Build an annotated path by including \(\phi_1, \ldots, \phi_n\) as assertions
2. Convert the path into SSA form
3. Replace assignments with \texttt{assume} over equality

\begin{align*}
\texttt{assert } & 0 \leq i \\
& i := 1 \\
\texttt{assert } & 0 \leq i \\
\texttt{assume } & 0 \leq x < 1 \\
\texttt{assert } & (0 \leq i) \\
& i := i - 1 \\
\texttt{assert } & 0 \leq i_0 \\
& i_1 := 1 \\
\texttt{assert } & 0 \leq i_1 \\
\texttt{assume } & 0 \leq x_0 < 1 \\
\texttt{assert } & (0 \leq i_1) \\
& i_0 := i_1 - 1
\end{align*}
Building Path Formulas

We’re given a path \((\ell_1, \phi_1), \ldots, (\ell_n, \phi_n)\)

1. Build an annotated path by including \(\phi_1, \ldots, \phi_n\) as assertions
2. Convert the path into SSA form
3. Replace assignments with **assume** over equality

\[
\begin{align*}
\text{assert } &0 \leq i \\
i &:= 1 \\
\text{assert } &0 \leq i \\
\text{assume } &0 \leq x < 1 \\
\text{assert } &\neg(0 \leq i) \\
i &:= i - 1 \\
\text{assert } &0 \leq i_0 \\
i_1 &:= 1 \\
\text{assert } &0 \leq i_1 \\
\text{assume } &0 \leq x_0 < 1 \\
\text{assert } &\neg(0 \leq i_1) \\
i_0 &:= i_1 - 1 \\
\text{assert } &0 \leq i_0 \\
\text{assume } &i_1 = 1 \\
\text{assert } &0 \leq i_1 \\
\text{assume } &0 \leq x_0 < 1 \\
\text{assume } &\neg(0 \leq i_1) \\
\text{assume } &i_2 = i_1 - 1 
\end{align*}
\]
We’re given a path \((\ell_1, \phi_1), \ldots, (\ell_n, \phi_n)\)

1. Build an annotated path by including \(\phi_1, \ldots, \phi_n\) as assertions
2. Convert the path into SSA form
3. Replace assignments with \textbf{assume} over equality
4. Compute weakest precondition of path wrt. \textit{true}

\begin{align*}
\text{assert } 0 \leq i \\
i := 1 \\
\text{assert } 0 \leq i \\
\text{assume } 0 \leq x < 1 \\
\text{assert } \neg (0 \leq i) \\
i := i - 1
\end{align*}

\begin{align*}
\text{assert } 0 \leq i_0 \\
i_1 := 1 \\
\text{assert } 0 \leq i_1 \\
\text{assume } 0 \leq x_0 < 1 \\
\text{assert } \neg (0 \leq i_1) \\
i_0 := i_1 - 1
\end{align*}

\begin{align*}
\text{assert } 0 \leq i_0 \\
\text{assume } i_1 = 1 \\
\text{assert } 0 \leq i_1 \\
\text{assume } 0 \leq x_0 < 1 \\
\text{assert } \neg (0 \leq i_1) \\
\text{assume } i_2 = i_1 - 1
\end{align*}
Building Path Formulas

We’re given a path \((\ell_1, \phi_1), \ldots, (\ell_n, \phi_n)\)

1. Build an annotated path by including \(\phi_1, \ldots, \phi_n\) as assertions
2. Convert the path into SSA form
3. Replace assignments with **assume** over equality
4. Compute weakest precondition of path wrt. **true**

\[
\begin{align*}
\text{assert } & 0 \leq i \\
& i := 1 \\
\text{assert } & 0 \leq i \\
\text{assume } & 0 \leq x < 1 \\
\text{assert } & \neg(0 \leq i) \\
& i := i - 1 \\
\text{assert } & 0 \leq i \\
& i_1 := 1 \\
\text{assert } & 0 \leq i_1 \\
\text{assume } & 0 \leq x_0 < 1 \\
\text{assert } & \neg(0 \leq i_1) \\
& i_0 := i_1 - 1 \\
\text{assert } & 0 \leq i \\
& i_2 := i_1 - 1 \\
\text{assume } & i_1 = 1 \\
\text{assume } & 0 \leq i_1 \\
\text{assume } & 0 \leq x_0 < 1 \\
\text{assume } & \neg(0 \leq i_1) \\
\text{assume } & i_2 = i_1 - 1 \\
\end{align*}
\]

\[wp(\ldots, \text{true}) = 0 \leq i_0 \land i_1 = 1 \land 0 \leq i_1 \land 0 \leq x_0 < 1 \land \neg(0 \leq i_1) \land i_2 = i_1 - 1\]
We have a counterexample, path formula pair
We have a counterexample, path formula pair

\[(i := 1, \text{true})\]
\[(\text{assume } 0 \leq x < 1, 0 \leq i)\]
\[(i := i - 1, \neg(0 \leq i))]\]
We have a counterexample, path formula pair

\[
(i := 1, \text{true})
\]

\[
\text{(assume } 0 \leq x < 1, 0 \leq i) \]

\[
(i := i - 1, \neg(0 \leq i))
\]

\[
0 \leq i_0 \land
i_1 = 1 \land
0 \leq i_1 \land
0 \leq x_0 < 1 \land
\neg(0 \leq i_1) \land
i_2 = i_1 - 1
\]

Is the path formula satisfiable?

No. We already knew this path was invalid.
We have a counterexample, path formula pair

\[(i := 1, \text{true})\]
\[(\text{assume } 0 \leq x < 1, 0 \leq i)\]
\[(i := i - 1, \neg(0 \leq i))\]

Is the path formula satisfiable?

**No.** We already knew this path was invalid
Learning New Predicates: Unsat Cores

How do we automatically find such an explanation?
Learning New Predicates: Unsat Cores

How do we automatically find such an explanation?
Learning New Predicates: Unsat Cores

How do we automatically find such an explanation?

**Recall:** An **unsatisfiable core** $C^*$ is a subset of $C$:
How do we automatically find such an explanation?

**Recall:** An **unsatisfiable core** $C^*$ is a subset of $C$:
- $C^*$ is still unsatisfiable
Learning New Predicates: Unsat Cores

How do we automatically find such an explanation?

**Recall:** An unsatisfiable core $C^*$ is a subset of $C$:
- $C^*$ is still unsatisfiable
- Dropping any element of $C^*$ makes it satisfiable
Learning New Predicates: Unsat Cores

How do we automatically find such an explanation?

**Recall:** An unsatisfiable core $C^*$ is a subset of $C$:
- $C^*$ is still unsatisfiable
- Dropping any element of $C^*$ makes it satisfiable

**To generate:** For each literal $l$ in $C$:

1. Drop $l$ from $C$ to build $C'$
2. If $C'$ is still unsatisfiable, then let $C^* = C'$
3. Otherwise, keep original $C$
How do we automatically find such an explanation?

**Recall:** An unsatisfiable core $C^*$ is a subset of $C$:
- $C^*$ is still unsatisfiable
- Dropping any element of $C^*$ makes it satisfiable

**To generate:** For each literal $l$ in $C$:
1. Drop $l$ from $C$ to build $C'$
Learning New Predicates: Unsat Cores

How do we automatically find such an explanation?

**Recall:** An unsatisfiable core $C^\ast$ is a subset of $C$:
- $C^\ast$ is still unsatisfiable
- Dropping any element of $C^\ast$ makes it satisfiable

**To generate:** For each literal $l$ in $C$:
1. Drop $l$ from $C$ to build $C'$
2. If $C'$ is still unsatisfiable, then let $C := C'$
How do we automatically find such an explanation?

**Recall:** An **unsatisfiable core** \( C^* \) is a subset of \( C \):
- \( C^* \) is still unsatisfiable
- Dropping any element of \( C^* \) makes it satisfiable

**To generate:** For each literal \( l \) in \( C \):
1. Drop \( l \) from \( C \) to build \( C' \)
2. If \( C' \) is still unsatisfiable, then let \( C := C' \)
3. Otherwise, keep original \( C \)
Learning New Predicates: Unsat Cores

How do we automatically find such an explanation?

Recall: An unsatisfiable core $C^*$ is a subset of $C$:
- $C^*$ is still unsatisfiable
- Dropping any element of $C^*$ makes it satisfiable

To generate: For each literal $l$ in $C$:
1. Drop $l$ from $C$ to build $C'$
2. If $C'$ is still unsatisfiable, then let $C := C'$
3. Otherwise, keep original $C$

We’ll modify this slightly:
How do we automatically find such an explanation?

**Recall:** An **unsatisfiable core** $C^*$ is a subset of $C$:
- $C^*$ is still unsatisfiable
- Dropping any element of $C^*$ makes it satisfiable

**To generate:** For each literal $l$ in $C$:
1. Drop $l$ from $C$ to build $C'$
2. If $C'$ is still unsatisfiable, then let $C := C'$
3. Otherwise, keep original $C$

We’ll modify this slightly:
1. First, enumerate every $l, l' \in C$ where $l \neq l'$
Learning New Predicates: Unsat Cores

How do we automatically find such an explanation?

**Recall:** An **unsatisfiable core** $C^\star$ is a subset of $C$:
- $C^\star$ is still unsatisfiable
- Dropping any element of $C^\star$ makes it satisfiable

**To generate:** For each literal $l$ in $C$:
1. Drop $l$ from $C$ to build $C'$
2. If $C'$ is still unsatisfiable, then let $C := C'$
3. Otherwise, keep original $C'$

We’ll modify this slightly:
1. First, enumerate every $l, l' \in C$ where $l \neq l'$
2. If $l \Rightarrow l'$, then remove $l'$
Example: Learning New Predicates

Initial formula:

\[ 0 \leq i_0 \land i_1 = 1 \land 0 \leq i_1 \land 0 \leq x_0 < 1 \land \neg(0 \leq i_1) \land i_2 = i_1 - 1 \]
Initial formula:

\[\begin{align*}
0 & \leq i_0 & \land \\
i_1 &= 1 & \land \\
0 & \leq i_1 & \land \\
0 & \leq x_0 < 1 & \land \\
\neg(0 & \leq i_1) & \land \\
i_2 &= i_1 - 1
\end{align*}\]

1: Remove \(0 \leq i_1 \ (i_1 = 1 \Rightarrow 0 \leq i_1)\)

\[\begin{align*}
0 & \leq i_0 & \land \\
i_1 &= 1 & \land \\
0 & \leq x_0 < 1 & \land \\
\neg(0 & \leq i_1) & \land \\
i_2 &= i_1 - 1
\end{align*}\]
Example: Learning New Predicates

Initial formula:

\[ 0 \leq i_0 \quad \land \\
\quad \land \\
\quad \land \\
\quad \land \\
\quad \land \\
\quad \land \\
\quad \land \\
\quad \land \\
\quad \land \\
\quad \land \]

\[ i_2 = i_1 - 1 \]

1: Remove \( 0 \leq i_1 \ (i_1 = 1 \Rightarrow 0 \leq i_1) \)

\[ 0 \leq i_0 \quad \land \\
\quad \land \\
\quad \land \\
\quad \land \\
\quad \land \\
\quad \land \\
\quad \land \\
\quad \land \\
\quad \land \\
\quad \land \]

\[ i_1 = 1 \quad \land \\
\quad \land \\
\quad \land \\
\quad \land \\
\quad \land \\
\quad \land \\
\quad \land \\
\quad \land \\
\quad \land \]

2: Remove \( 0 \leq i_0 \)

\[ i_1 = 1 \quad \land \\
\quad \land \\
\quad \land \\
\quad \land \\
\quad \land \\
\quad \land \\
\quad \land \\
\quad \land \\
\quad \land \]

\[ 0 \leq x_0 < 1 \quad \land \\
\quad \land \\
\quad \land \\
\quad \land \\
\quad \land \\
\quad \land \\
\quad \land \\
\quad \land \]

\[ \neg (0 \leq i_1) \quad \land \\
\quad \land \\
\quad \land \\
\quad \land \\
\quad \land \\
\quad \land \\
\quad \land \]

\[ i_2 = i_1 - 1 \]

\[ i_2 = i_1 - 1 \]

wasn't previously in our set, so we refine by adding it
Example: Learning New Predicates

Initial formula:

1: Remove $0 \leq i_1 \ (i_1 = 1 \Rightarrow 0 \leq i_1)$

- $0 \leq i_0$
- $i_1 = 1$
- $0 \leq i_1$
- $0 \leq x_0 < 1$
- $\neg (0 \leq i_1)$
- $i_2 = i_1 - 1$

2: Remove $0 \leq i_0$

- $i_1 = 1$
- $0 \leq x_0 < 1$
- $\neg (0 \leq i_1)$
- $i_2 = i_1 - 1$

3: Remove $0 \leq x_0 < 1$

- $i_1 = 1$
- $\neg (0 \leq i_1)$
- $i_2 = i_1 - 1$
Example: Learning New Predicates

Initial formula:

1: Remove $0 \leq i_1 \ (i_1 = 1 \Rightarrow 0 \leq i_1)$

- $0 \leq i_1$
- $i_1 = 1$
- $0 \leq i_1$
- $0 \leq x_0 < 1$
- $\neg (0 \leq i_1)$
- $i_2 = i_1 - 1$

2: Remove $0 \leq i_0$

- $0 \leq i_0$
- $i_1 = 1$
- $0 \leq x_0 < 1$
- $\neg (0 \leq i_1)$
- $i_2 = i_1 - 1$

3: Remove $0 \leq x_0 < 1$

- $i_1 = 1$
- $\neg (0 \leq i_1)$
- $i_2 = i_1 - 1$

4: Remove $i_2 = i_1 - 1$

- $i_1 = 1$
- $\neg (0 \leq i_1)$
Example: Learning New Predicates

Initial formula:

$0 \leq i_0 \land i_1 = 1 \land 0 \leq i_1 \land 0 \leq x_0 < 1 \land \neg(0 \leq i_1) \land i_2 = i_1 - 1$

1: Remove $0 \leq i_1$ \(i_1 = 1 \Rightarrow 0 \leq i_1\)

$0 \leq i_0 \land i_1 = 1 \land 0 \leq x_0 < 1 \land \neg(0 \leq i_1) \land i_2 = i_1 - 1$

2: Remove $0 \leq i_0$

$i_1 = 1 \land 0 \leq x_0 < 1 \land \neg(0 \leq i_1) \land i_2 = i_1 - 1$

3: Remove $0 \leq x_0 < 1$

$i_1 = 1 \land \neg(0 \leq i_1) \land i_2 = i_1 - 1$

4: Remove $i_2 = i_1 - 1$

$i_1 = 1 \land \neg(0 \leq i_1) \land i_2 = i_1 - 1$

$i_1 = 1$ wasn’t previously in our set, so we refine by adding it
Path formulas give us a way to check invariants on individual paths.
Path formulas give us a way to check invariants on individual paths

1. Given an invariant property $G \phi$, 

$\begin{align*}
&\text{Enumerate a sequence of statements } \ell_1; \ldots; \ell_n \\
&\text{Create the "counterexample" } (\ell_1; \text{true}); \ldots; (\ell_n; \text{true}); (\text{skip}; : \phi) \\
&\text{Generate the path formula } \phi_{\text{path}} \\
&\text{Check } \phi_{\text{path}} \text{ for satisfiability}
\end{align*}$

The results tell us everything:

- If $\text{unsat}$, there's no way to execute $\ell_1; \ldots; \ell_n$ satisfying $\phi$
- If $\text{sat}$, then this path is a valid counterexample
- A $\text{sat}$ assignment to initial SSA variables is an input to the program
- When run on these inputs, the property will be violated
Path formulas give us a way to check invariants on individual paths

1. Given an invariant property $G \phi$,
2. Enumerate a sequence of statements $\ell_1, \ldots, \ell_n$
Bounded Model Checking

Path formulas give us a way to check invariants on individual paths

1. Given an invariant property $\mathbf{G} \phi$,
2. Enumerate a sequence of statements $\ell_1, \ldots, \ell_n$
3. Create the “counterexample” $(\ell_1, \text{true}), \ldots, (\ell_n, \text{true}), (\text{skip}, \neg \phi)$
Path formulas give us a way to check invariants on individual paths

1. Given an invariant property $\mathbf{G} \, \phi$,
2. Enumerate a sequence of statements $\ell_1, \ldots, \ell_n$
3. Create the “counterexample” $(\ell_1, true), \ldots, (\ell_n, true), (\text{skip}, \neg \phi)$
4. Generate the path formula $\phi_{\text{path}}$
Path formulas give us a way to check invariants on individual paths

1. Given an invariant property \( G \phi \),
2. Enumerate a sequence of statements \( \ell_1, \ldots, \ell_n \)
3. Create the “counterexample” \((\ell_1, true), \ldots, (\ell_n, true), (\text{skip}, \neg \phi)\)
4. Generate the path formula \( \phi_{\text{path}} \)
5. Check \( \phi_{\text{path}} \) for satisfiability
Path formulas give us a way to check invariants on individual paths

1. Given an invariant property $\mathbf{G} \phi$,
2. Enumerate a sequence of statements $\ell_1, \ldots, \ell_n$
3. Create the “counterexample” $(\ell_1, \text{true}), \ldots, (\ell_n, \text{true}), (\text{skip}, \neg \phi)$
4. Generate the path formula $\phi_{\text{path}}$
5. Check $\phi_{\text{path}}$ for satisfiability

The results tell us everything:
Path formulas give us a way to check invariants on individual paths

1. Given an invariant property $\mathbf{G} \phi$, 
2. Enumerate a sequence of statements $\ell_1, \ldots, \ell_n$ 
3. Create the “counterexample” $(\ell_1, \text{true}), \ldots, (\ell_n, \text{true}), (\text{skip}, \neg \phi)$ 
4. Generate the path formula $\phi_{\text{path}}$ 
5. Check $\phi_{\text{path}}$ for satisfiability

The results tell us everything:

- If *unsat*, there’s no way to execute $\ell_1, \ldots, \ell_n$ satisfying $\neg \phi$
Path formulas give us a way to check invariants on individual paths

1. Given an invariant property $\mathbf{G} \ \phi$, 
2. Enumerate a sequence of statements $\ell_1, \ldots, \ell_n$
3. Create the “counterexample” $(\ell_1, \text{true}), \ldots, (\ell_n, \text{true}), (\text{skip}, \neg \phi)$
4. Generate the path formula $\phi_{\text{path}}$
5. Check $\phi_{\text{path}}$ for satisfiability

The results tell us everything:

- If $\text{unsat}$, there’s no way to execute $\ell_1, \ldots, \ell_n$ satisfying $\neg \phi$
- If $\text{sat}$, then this path is a valid counterexample
Path formulas give us a way to check invariants on individual paths

1. Given an invariant property $G \phi$,
2. Enumerate a sequence of statements $\ell_1, \ldots, \ell_n$
3. Create the “counterexample” $(\ell_1, true), \ldots, (\ell_n, true), (\text{skip}, \neg \phi)$
4. Generate the path formula $\phi_{\text{path}}$
5. Check $\phi_{\text{path}}$ for satisfiability

The results tell us everything:

- If $\text{unsat}$, there’s no way to execute $\ell_1, \ldots, \ell_n$ satisfying $\neg \phi$
- If $\text{sat}$, then this path is a valid counterexample

$sat$ assignment to initial SSA variables is an input to the program
Path formulas give us a way to check invariants on individual paths

1. Given an invariant property $G \phi$,
2. Enumerate a sequence of statements $\ell_1, \ldots, \ell_n$
3. Create the “counterexample” $(\ell_1, true), \ldots, (\ell_n, true), (\text{skip}, \neg \phi)$
4. Generate the path formula $\phi_{\text{path}}$
5. Check $\phi_{\text{path}}$ for satisfiability

The results tell us everything:

- If unsat, there’s no way to execute $\ell_1, \ldots, \ell_n$ satisfying $\neg \phi$
- If sat, then this path is a valid counterexample

sat assignment to initial SSA variables is an input to the program
- When run on these inputs, the property will be violated
Bounded Model Checking: Example

\[ \ell_0 : i := 1; \]
\[ \ell_1 : \textbf{while} (0 \leq x < 2) \{ \]
\[ \ell_2 : i := i - 1; \]
\[ \ell_3 : x := x + 1; \]
\[ \} \]
\[ \ell_e : \textbf{assert} (0 \leq i) \]
Bounded Model Checking: Example

\[ \ell_0 : i := 1; \]
\[ \ell_1 : \textbf{while}(0 \leq x < 2) \{ \]
\[ \ell_2 : i := i - 1; \]
\[ \ell_3 : x := x + 1; \]
\[ \} \]
\[ \ell_e : \textbf{assert}(0 \leq i) \]

We suspect the path:

\[ i := 1; \]
\[ \textbf{assume}(0 \leq x < 2) \]
\[ i := i - 1; \]
\[ x := x + 1; \]
\[ \textbf{assume}(0 \leq x < 2) \]
\[ i := i - 1; \]
\[ x := x + 1; \]
\[ \textbf{assume}(\neg(0 \leq x < 2)) \]
\[ \textbf{assert}(0 \leq i) \]
Bounded Model Checking: Example

\[ l_0 : i := 1; \]
\[ l_1 : \textbf{while}(0 \leq x < 2) \{ \]
\[ l_2 : i := i - 1; \]
\[ l_3 : x := x + 1; \]
\[ \} \]
\[ l_e : \textbf{assert}(0 \leq i) \]

We suspect the path:

\[ i := 1; \]
\[ \textbf{assume}(0 \leq x < 2) \]
\[ i := i - 1; \]
\[ x := x + 1; \]
\[ \textbf{assume}(0 \leq x < 2) \]
\[ i := i - 1; \]
\[ x := x + 1; \]
\[ \textbf{assume}(\neg(0 \leq x < 2)) \]
\[ \textbf{assert}(0 \leq i) \]

After SSA, assumption encoding:

\[ \textbf{assume} \ i_1 = 1; \]
\[ \textbf{assume} \ 0 \leq x_0 < 2; \]
\[ \textbf{assume} \ i_2 = i_1 - 1; \]
\[ \textbf{assume} \ x_1 = x_0 + 1; \]
\[ \textbf{assume} \ 0 \leq x_1 < 2; \]
\[ \textbf{assume} \ i_3 = i_2 - 1; \]
\[ \textbf{assume} \ x_2 = x_1 + 1; \]
\[ \textbf{assume} \ \neg(0 \leq x_2 < 2); \]
\[ \textbf{assert} \ 0 \leq i_3; \]
Bounded Model Checking: Example

\[ \ell_0 : i := 1; \]
\[ \ell_1 : \text{while}(0 \leq x < 2) \{ \]
\[ \ell_2 : i := i - 1; \]
\[ \ell_3 : x := x + 1; \]
\[ \} \]
\[ \ell_e : \text{assert}(0 \leq i) \]

Path formula:

\[ i_1 = 1 \wedge \]
\[ 0 \leq x_0 < 2 \wedge \]
\[ i_2 = i_1 - 1 \wedge \]
\[ x_1 = x_0 + 1 \wedge \]
\[ 0 \leq x_1 < 2 \wedge \]
\[ i_3 = i_2 - 1 \wedge \]
\[ x_2 = x_1 + 1 \wedge \]
\[ \neg(0 \leq x_2 < 2) \wedge \]
\[ 0 \leq i_3 \]

We suspect the path:

\[ i := 1; \]
\[ \text{assume}(0 \leq x < 2) \]
\[ i := i - 1; \]
\[ x := x + 1; \]
\[ \text{assume}(0 \leq x < 2) \]
\[ i := i - 1; \]
\[ x := x + 1; \]
\[ \text{assume}(\neg(0 \leq x < 2)) \]
\[ \text{assert}(0 \leq i) \]
Bounded Model Checking: Example

\[
\begin{align*}
\ell_0 &: i := 1; \\
\ell_1 &: \text{while}(0 \leq x < 2) \{ \\
\ell_2 &: i := i - 1; \\
\ell_3 &: x := x + 1; \\
\} \\
\ell_e &: \text{assert}(0 \leq i)
\end{align*}
\]

Path formula:

\[
\begin{align*}
i_1 &= 1 \\
0 &\leq x_0 < 2 \\
i_2 &= i_1 - 1 \\
x_1 &= x_0 + 1 \\
0 &\leq x_1 < 2 \\
i_3 &= i_2 - 1 \\
x_2 &= x_1 + 1 \\
\neg(0 &\leq x_2 < 2) \\
0 &\leq i_3
\end{align*}
\]

Is this satisfiable?

We suspect the path:

\[
\begin{align*}
i &= 1; \\
\text{assume}(0 \leq x < 2) \\
i &= i - 1; \\
x &= x + 1; \\
\text{assume}(0 \leq x < 2) \\
i &= i - 1; \\
x &= x + 1; \\
\text{assume}(\neg(0 \leq x < 2)) \\
\text{assert}(0 \leq i)
\end{align*}
\]
Bounded Model Checking: Example

\( \ell_0 : i := 1; \)
\( \ell_1 : \textbf{while}(0 \leq x < 2) \{ \)
\( \ell_2 : i := i - 1; \)
\( \ell_3 : x := x + 1; \)
\( \} \)
\( \ell_e : \textbf{assert}(0 \leq i) \)

We suspect the path:

\( i := 1; \)
\( \textbf{assume}(0 \leq x < 2) \)
\( i := i - 1; \)
\( x := x + 1; \)
\( \textbf{assume}(0 \leq x < 2) \)
\( i := i - 1; \)
\( x := x + 1; \)
\( \textbf{assume}(\neg(0 \leq x < 2)) \)
\( \textbf{assert}(0 \leq i) \)

\[ \text{Path formula:} \]
\[ i_1 = 1 \quad \land \]
\[ 0 \leq x_0 < 2 \quad \land \]
\[ i_2 = i_1 - 1 \quad \land \]
\[ x_1 = x_0 + 1 \quad \land \]
\[ 0 \leq x_1 < 2 \quad \land \]
\[ i_3 = i_2 - 1 \quad \land \]
\[ x_2 = x_1 + 1 \quad \land \]
\[ \neg(0 \leq x_2 < 2) \quad \land \]
\[ 0 \leq i_3 \]

Is this satisfiable?
\[ i_1 = 1, x_0 = 0, i_2 = 0, x_1 = 1, i_3 = -1, x_2 = 2 \]
Bounded Model Checking: Example

\[ \ell_0 : i := 1; \]
\[ \ell_1 : \text{while}(0 \leq x < 2) \{ \]
\[ \ell_2 : i := i - 1; \]
\[ \ell_3 : x := x + 1; \]
\[ \} \]
\[ \ell_e : \text{assert}(0 \leq i) \]

We suspect the path:

\[ i := 1; \]
\[ \text{assume}(0 \leq x < 2) \]
\[ i := i - 1; \]
\[ x := x + 1; \]
\[ \text{assume}(0 \leq x < 2) \]
\[ i := i - 1; \]
\[ x := x + 1; \]
\[ \text{assume}(\neg(0 \leq x < 2)) \]
\[ \text{assert}(0 \leq i) \]

Path formula:

\[ i_1 = 1 \]
\[ 0 \leq x_0 < 2 \]
\[ i_2 = i_1 - 1 \]
\[ x_1 = x_0 + 1 \]
\[ 0 \leq x_1 < 2 \]
\[ i_3 = i_2 - 1 \]
\[ x_2 = x_1 + 1 \]
\[ \neg(0 \leq x_2 < 2) \]
\[ 0 \leq i_3 \]

Is this satisfiable?

\[ i_1 = 1, x_0 = 0, i_2 = 0, x_1 = 1, i_3 = -1, x_2 = 2 \]

We can use \( x = 0 \) as an initial test case.
Next Lecture

Go over homeworks
Go over homeworks

Review for the final
Next Lecture

Go over homeworks

Review for the final

Last homework due on Friday evening, 11:59
  ▶ No late days!
  ▶ University policy...